Comparing different versions of SDBot using SABRE BinDiff v1.7

Sebastian Porst, September 2005

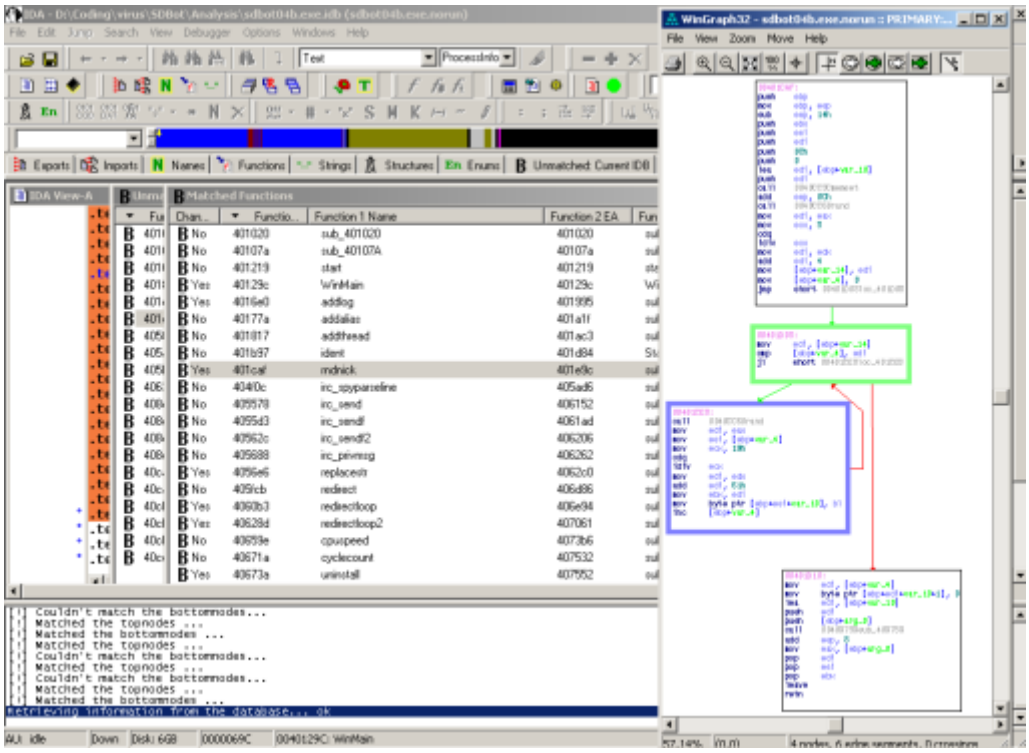

Table of Contents

1. Introduction

In the last few years the form and behaviour of malware changed significantly ([Mashevsky2005]). The ever growing popularity of the Internet led to a decline of the traditional viruses which spread and worked independently from the virus creator once they were released into the wild. This old form of malware was slowly replaced by Trojan horses and bots which feature advanced remote control options that allow their creator to control their spread and infection process or to update the program on the fly.

This new form of malware is significantly more complex than those of the past. Among other things modern malware often feature different ways to spread (ICQ, AIM, IRC, ...), they include recently published operating-system or application exploits of all sorts to maximize the number of infected machines and they contain entire servers for technologies like FTP or the e-mail protocols.
One example of an extremely complex piece of malware is the so called Reptile Bot. The source code of this bot contains more than 200 files organized in several Visual C++ projects making the bot as complex as an average shareware or freeware program.

This increasing complexity reached its logical conclusion when new pieces of malware started to follow versioning and upgrading rules that have been commonplace in “real” software projects for decades. When a malicious programmer learns about the shortcomings of earlier versions of his Trojan horse he merely creates updates that fix those problems instead of writing a new bot. Coupled with the fact that many of these pieces of malware are open-source and readily available on the Internet for everyone to use and abuse it's obvious that lots of different versions of the same piece of malware exist.

An extreme example of this is a bot called SDBot which this paper focuses on. The virus database of Sophos Antivirus claims to know about 1300 different versions of this bot[1]. Now that number is certainly exaggerated because two different compilations of the same bot that only differ in some connection or authentication information are for all intents and purposes still the same bot. Nevertheless assuming the existence of several dozen different versions of SDBot is probably realistic.

Next to the undisputedly malicious programs already mentioned a new category of malware surfaced in the last years: Spyware, according to Wikipedia[2] “a broad category of malicious software intended to intercept or take partial control of a computer's operation without the user's informed consent”.
Even though many spyware creators pretend to be legitimate businesses Anti-Virus companies started to blacklist their software. If that's justified or not is not up to me to decide but I can point out the similarities between spyware and other pieces of modern malware that are relevant for this paper. As spyware applications pretend to be legitimate applications they generally follow the rules of proper software development both in terms of application complexity and in terms of upgrading and versioning.

These new developments in malware creation call for new tools for examining malware that allow anti-virus researchers to react quickly when a new version of a known piece of

1 http://www.sophos.com/search/index.cgi?scope=whole_site&terms=sdbot&x=0&y=0
2 http://en.wikipedia.org/wiki/Spyware

malware hits the wild. This paper is about one of these new tools: SABRE BinDiff

2. An overview of BinDiff

BinDiff is a relatively new tool which has matured to version 1.7 as of the time I started to write this paper in September 2005. It was developed by Halvar Flake and Rolf Rolles of SABRE Security[3]. Working as a plugin for the popular disassembler IDA Pro[4] 4.8 (Interactive Disassembler Pro) it compares two executable files and attempts to match functions between the two files even when the code inside the functions or the relative order of the functions in the file changed.

To accomplish this the researchers at SABRE Security created sophisticated algorithms based on graph theory and the search for so called fixpoints, points that uniquely identify a chunk of code, in executable files. There's no need to go further into the details of this process in this paper as the SABRE Security website hosts two excellent papers that explain how everything works. If you're interested in the mathematical and theoretical background of matching pieces of code between binary files I suggest you read [FlakeDimva2004] and [BinDiffSSTIC05].

Once the user asked BinDiff to compare two IDB databases (the database format IDA uses) he has several options:
By default BinDiff displays the matched and unmatched functions using a simple grid that allows the user to quickly navigate from function to function.
Furthermore the tool can display flow charts of the matched functions. This makes it very easy for the user to verify the results BinDiff created as the flowcharts of two different versions of the same function should still be relatively similar.
BinDiff eases the process of recognizing matches and differences within two matched functions even more by colouring the nodes in the chart in different ways but more about that later.
Last but not least BinDiff can automatically port function names, anterior and posterior comment lines, standard comments, and local names from one disassembly to the other. The use of this powerful and time-saving option increases with the size of the executables you're working with and with the number of comments in the IDB databases.

More about all these options will be presented later in this paper.

3. An overview of SDBot

To demonstrate the usefulness of BinDiff in malware analysis it's necessary to provide an example. I've decided to use a Trojan Horse called SDBot written by the Russian virus programmer [sd] for this purpose.

There are three reasons for my decision:

- I already have access to several versions of SDBot.
- SDBot is a relatively simple bot, it's source code is just a single C file.
- SDBot exists in lots of different versions and is therefore a prime candidate for BinDiff

3 http://www.sabre-security.com/
4 http://www.datarescue.com/

To allow readers to fully understand the results I'm about to present in the next part of this paper I want to give a short overview of the functionality of SDBot now. Like I said, SDBot is rather small. Depending on the exact version of the bot its single source file is between 50 KB and 70 KB large. The code of the bot is structured in 25 – 30 different functions. Symantec describes the functionality of the bot the following way[5]: "*Backdoor.Sdbot is a Trojan horse that opens a back door and allows a remote attacker to control a computer by using Internet Relay Chat (IRC). The Trojan can update itself by checking for newer versions on the Internet.* " The current Symantec threat assessment of this bot can be seen in illustration 1.

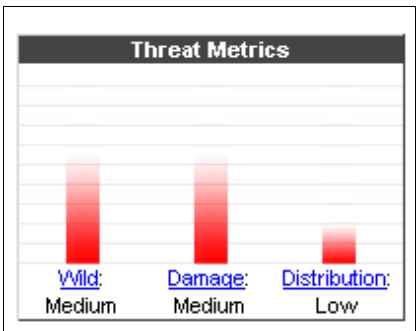

Illustration 1: SDBot threat metrics

4. Working with BinDiff

Before drawing any conclusions I want to use this part of the paper to explain what working with BinDiff is like. To explain how BinDiff can be used to simplify malware analysis I set up the following experiment. I took a number of different versions of SDBot executables and starting with the earliest version I have I used BinDiff to speed up analysis of the other versions. The following four different versions of SDBot are used for this purpose:

- SDBot 0.4b I compiled myself with lcc-win32 3.8
- SDBot 0.5a I compiled myself with lcc-win32 3.8
- SDBot 0.5b I compiled myself with lcc-win32 3.8
- SDBot 0.5a I found in the wild compiled with Visual C++ 6

The very first step that's necessary to benefit from BinDiff is to analyze and annotate the first version of the malware you find. This is a standard procedure when analyzing malware and happens before the anti-virus researcher even knows for sure that different versions of the malware will spread in the future.

In the example scenario this means disassembling the executable file of SDBot 0.4b in IDA and assigning names to the functions. As SDBot is open-source I've decided to use the function names from the original source code instead of creating new ones. This first executable file contains 185 functions of which only 26 are actual SDBot code. The other functions are part of the C standard library or whatever else lcc adds to the binary file. Only the 26 SDBot functions and how successfully BinDiff matches them are examined further in this paper.

After starting BinDiff and selecting the appropriate files to compare, BinDiff works for a few seconds and opens three new windows in IDA when it's done. The first of these three windows gives information about the functions BinDiff could match between the two files.

5 http://securityresponse.symantec.com/avcenter/venc/data/backdoor.sdbot.html

The other two windows show the functions which couldn't be matched in the first and second file.

Depending on what exactly you're doing, the matched or the unmatched functions might be more important. Nevertheless it's a good idea to start working with the unmatched functions because even though BinDiff is already very good it's not omniscient and not all equal functions can be matched between two binary files. Some manual work is still necessary.

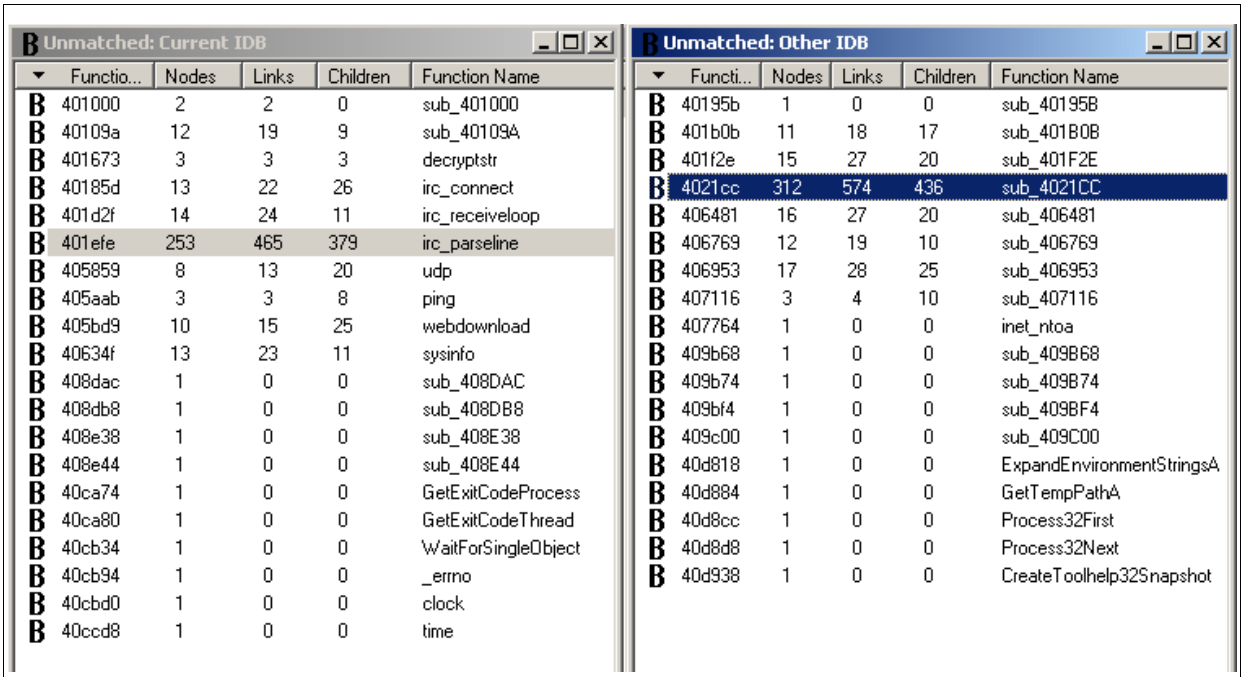

Illustration 2: Unmatched Functions

In Illustration 2 you can see the two windows showing information about the unmatched functions. In the "*Unmatched Other*" window the user has the option to match functions manually. To do so he right-clicks onto a function and chooses the option "*View probable matches*" from the context-menu. Doing so makes a dialog pop up which allows the user to choose from a list of functions that are likely matches of the selected function (Illustration 4). In the example I've opted for the function irc_parseline, the biggest function in SDBot, which can easily be matched manually because its large number of nodes, links and children stands out compared to the same numbers of other functions.

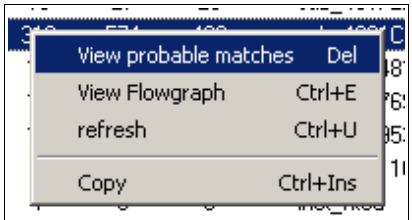

Illustration 3: Unmatched Functions context menu

If BinDiff suggests several potential matches for a function and the user is still unsure which function is the real match there's also the option to view the flow graph of the two functions. As the flow graphs and especially the code of two different versions of the same function is in most cases remarkably similar, comparing the flow graphs removes the last ambiguity most of the time.

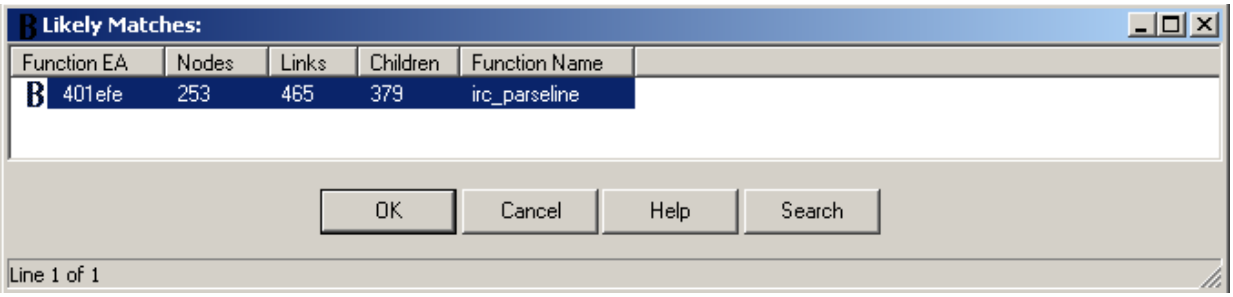

Illustration 4: Likely Matches

After the user finished matching unmatched functions manually he can now move on to the matched functions. The "*Matched Functions*" window (Illustration 5) contains a grid with five columns. The first column says whether the function changed between the two versions. The other four columns give the name and the effective address of the matched functions. At this point the user has several options which are provided through the context menu you can see in Illustration 6.

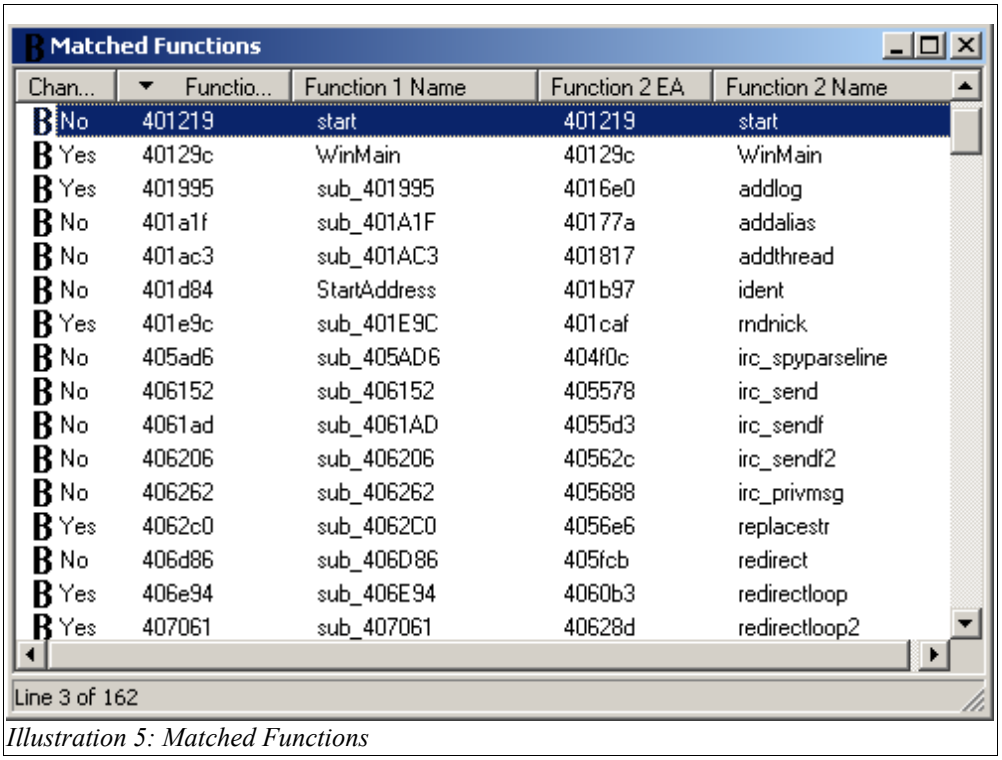

Illustration 5: Matched Functions

The first option, "*Port*", copies all sorts of comments and function names from the first file to the second one. How exactly this works can be configured in the BinDiff Options dialog. It's important to note at this point that this option will port the comments and names of all functions from one file to the other, not just the name and comments of the selected function.

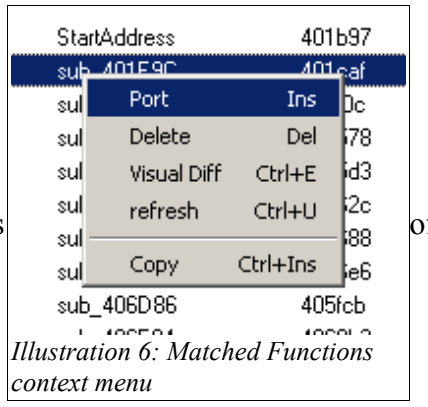

Illustration 6: Matched Functions context menu

The flow graph option for matched functions is a bit different to the one for unmatched functions because it's now possible to highlight differences and similarities

between the two functions directly in the flow chart. In Illustration 7 you can see what BinDiff has to offer after chosing the "*Visual Diff*" option for the rndnick function, a function that generates a random nick to be used after the bot connects to an IRC server.

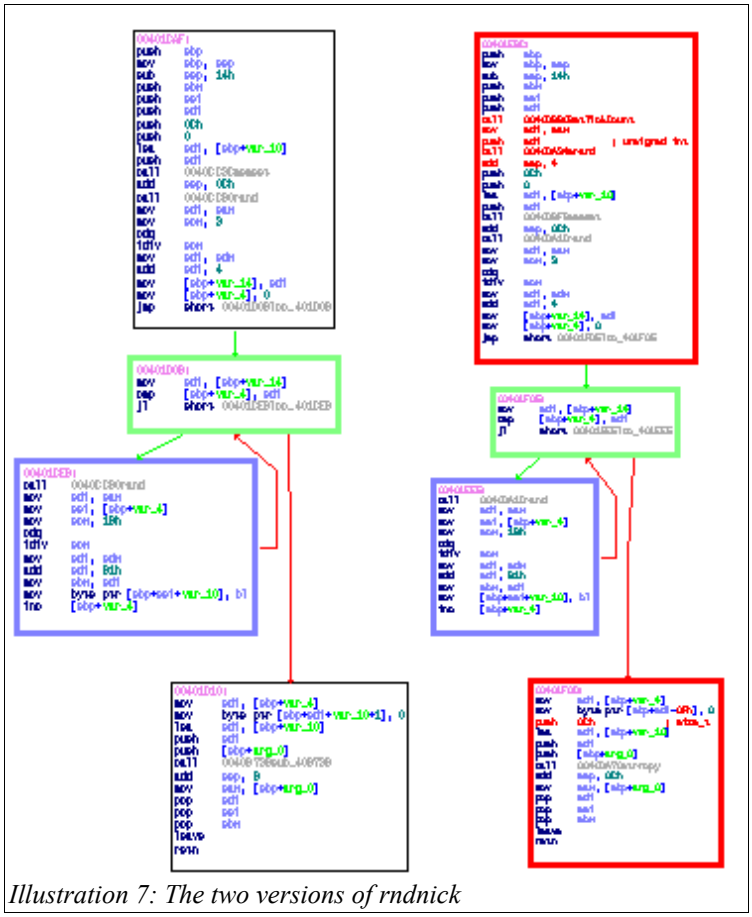

Illustration 7: The two versions of rndnick

How and if node and code changes are highlighted in the flow graph can be configured in the BinDiff options dialog. In the example, a red node border means that the node changed between the two versions and red code means that code changed between the two versions. Green borders mark the entry node of loops while blue borders mark the exit block of loops.

In Illustration 7 you can see that the first and the last node of the function changed. Giving the first node a closer look (Illustration 8; left side) you can notice that code was added to initialize the C random number generator (srand) with a seed calculated from the Windows API function GetTickCount. This part was previously missing. The difference in the last node (illustration 8; right side) is that the bot moved from strcpy to the safer strncpy when copying the generated nick from one string buffer to another one.

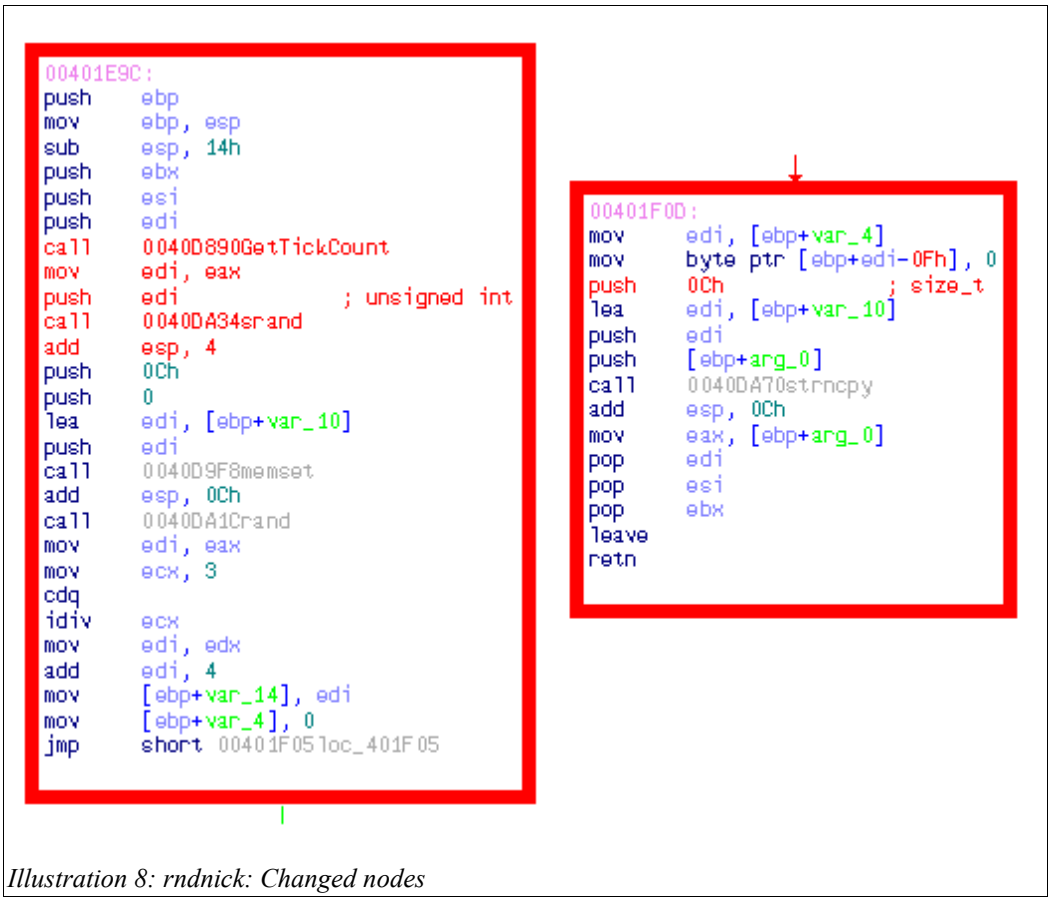

Illustration 8: rndnick: Changed nodes

5. Conclusions

After the process of working with BinDiff has now been thoroughly explained it's time to look at the actual results of my SDBot experiment. I was quite happy with BinDiff and the numbers I'm about to present serve to illustrate my point.

The following table shows the recognition rate of BinDiff when comparing different versions of SDBot with each other. Each version of SDBot was compared with the one before. The exception is 0.5a VC++ which was compared with the SDBot version 0.5a LCC. The high matching rate between the same source file compiled with two different compilers was surprising.

The column *Matched (A)* shows the number of functions BinDiff automatically matched correctly.
Matched (M) are the number of functions BinDiff didn't match automatically but provided a single, correct likely match for me to match the functions manually.
The third column, *Matched (I)* contains the number of functions I was able to match after applying the manual matching process iteratively. That means I went ahead and manually matched all functions for which BinDiff suggested a single, correct likely match. Doing that improves the result of other functions by correcting formerly ambigious results. I continued that until only incorrect or ambigious matches were left.
The number of functions BinDiff couldn't match without the user having a good look at the

function code can be found in the last column. Note that there are new functions in newer versions of SDBot and BinDiff obviously can't find any matches for them. That means the number in the last column is not equivalent to the number of incorrect matches BinDiff made.

SDBot	*Filesize*	*Functions*	*Matched (A)*	*Matched (M)*	*Matched (I)*	*Unmatched*
0.4b (LCC)	59,424	26	-	-	-	-
0.5a (LCC)	63,520	27	18	3	3	3
0.5b (LCC)	61,984	28	21	3	0	4
0.5a (VC++)	199,168	27	22	1	0	4

6. References

[Mashevsky2005] - Yury Mashevsky; Watershed in malicious code evolution; July 29 2005; http://www.viruslist.com/en/viruses/analysis?pubid=167798878

[FlakeDimva2004] – Halvar Flake; Structural Comparison of Executable Objects; 2004; http://www.sabre-security.com/files/dimva_paper2.pdf

[BinDiffSSTIC05] – Thomas Dullien, Rolf Rolles; Graph-based comparison of executable objects; 2005; http://www.sabre-security.com/files/BinDiffSSTIC05.pdf